Catching the Bugs You're Missing

Kinds of Testing

Example-Based Tests

```
expect(1 + 2).to eq(3)
expect(1 + 2).to eq(3)
expect(1 + 2).to eq(3)
expect(add(1, 2)).to eq(3)
expect(add(1, 2)).to eq(3)
expect(add(2, 1)).to eq(3)
expect(add(1, 2)).to eq(3)
expect(add(2, 1)).to eq(3)
expect(add(1, 0)).to eq(1)
expect(add(1, 2)).to eq(3)
expect(add(2, 1)).to eq(3)
expect(add(1, 0)).to eq(1)
expect(add(0, 1)).to eq(1)
expect(add(1, 2)).to eq(3)
expect(add(2, 1)).to eq(3)
expect(add(1, 0)).to eq(1)
expect(add(0, 1)).to eq(1)
expect(add(-1, 1)).to eq(0)
expect(add(1, 2)).to eq(3)
expect(add(2, 1)).to eq(3)
expect(add(1, 0)).to eq(1)
expect(add(0, 1)).to eq(1)
expect(add(-1, 1)).to eq(0)
expect(add(1, -1)).to eq(0)
```

Property-Based Tests

```
# For some integer x
#       and integer y

         add(x,y)
      == add(y,x)
```

```
# For some integer x
#       and integer y

expect(add(x,y))
.to eq(add(y,x))
```

```
property_of {
  [integer, integer]
}.check { |x,y|
  expect(add(x,y))
  .to eq(add(y,x))
}
```

```
it "has swappable args" do
    property_of {
      [integer, integer]
    }.check { |x,y|
      expect(add(x,y))
      .to eq(add(y,x))
    }
end
```

```
# For some integer x

         add(x,0)
      == x
```

```
it "has a do-nothing val" do
    property_of {
      integer
    }.check { |x|
      expect(add(x,0))
      .to eq(x)
    }
end
```

```
# For some integer x

   add(x,x)
== x * 2
```

```
it "matches multiplic'n" do
    property_of {
      integer
    }.check { |x|
      expect(add(x,x))
      .to eq(x * 2)
    }
end
```

Something that breaks

For some string, x:

```
x.split(" ").join(" ") == x
```

```
it "split/join is reversible" do
  property_of {
    string
  }.check { |x|
    expect(
      x.split(" ").join(" ")
    ).to eq(x)
  }
end
```

```
failure after 7 tests, on:
" &2M1`"
found a reduced failure case:
" &M1`"
found a reduced failure case:
" M1`"
found a reduced success:
"M1`"
minimal failed data is:
" M1`"
  split/join is reversible (FAILED - 1)
```

```
failure after 7 tests, on:
" &2M1`"
found a reduced failure case:
" &M1`"
found a reduced failure case:
" M1`"
found a reduced success:
"M1`"
minimal failed data is:
" M1`"
  split/join is reversible (FAILED - 1)
```

```
failure after 7 tests, on:
" &2M1`"
found a reduced failure case:
" &M1`"
found a reduced failure case:
" M1`"
found a reduced success:
"M1`"
minimal failed data is:
" M1`"
  split/join is reversible (FAILED - 1)
```

```
failure after 7 tests, on:
" &2M1`"
found a reduced failure case:
" &M1`"
found a reduced failure case:
" M1`"
found a reduced success:
"M1`"
minimal failed data is:
" M1`"
  split/join is reversible (FAILED - 1)
```

```
failure after 7 tests, on:
" &2M1`"
found a reduced failure case:
" &M1`"
found a reduced failure case:
" M1`"
found a reduced success:
"M1`"
minimal failed data is:
" M1`"
  split/join is reversible (FAILED - 1)
```

Three Things:

Data Generation, Testing with the Data, Data Reduction

Uses

Edge Cases

Treating the code like an adversary

Honest TDD;
No fudging code to
pass an example test.

Kinds of Properties

Reversible

Reversible

```
n + 1 - 1 == n
```

Reversible

```
n + 1 - 1 == n
```

```
# where y != " "
x.split(y).join(y) == x
```

Reversible

```
n + 1 - 1 == n

# where y != " "
x.split(y).join(y) == x

decompress(compress(d)) == d
```

Repeatable

Repeatable

```
list.sort.sort == list.sort
```

Repeatable

```
list.sort.sort == list.sort

  Foo.new(attr).tap(&:save)
     .attributes
   ==
  Foo.new(attr).tap(&:save)
     .tap(&:save)
     .attributes
```

Unbreakable Rules

Unbreakable Rules

```
list.sort.count == list.count
```

Unbreakable Rules

```
list.sort.count == list.count

list.sort.all? {|x|
  list.find_index(x) != nil
}
```

Prove a Small Part

Prove a Small Part

```
pairs(list.sort).all?{|(x,y)|
  x <= y
}

# pairs([1,2,3])
#  => [[1,2], [2,3]]
```

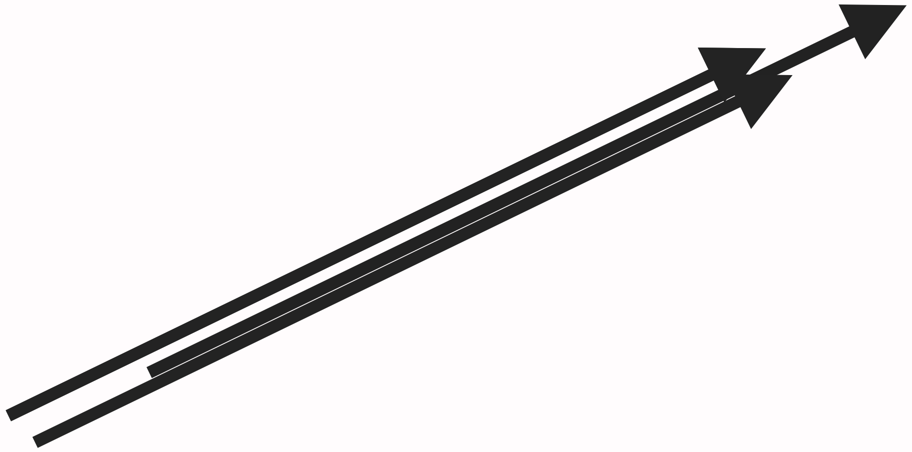
Swapping the Ordering

Swapping the Ordering

```
a.map{|n| n + 1}.sort
         ==
a.sort.map{|n| n + 1}
```

Hard to Solve, Easy to Check

Hard to Solve, Easy to Check



Consult an Oracle

Consult an Oracle

```
list.hypersort == list.sort
```

Consult an Oracle

```
list.hypersort == list.sort
```

```
new_code(input) == old_code(input)
```

SHOW US SOME REAL EXAMPLES

```
it "can round-trip last-logged-in" do
  property_of {
    (Time.current - float.abs)
  }.check { |time|
    user = User.create(
      username: "Sam",
      last_logged_in_at: time,
    )
    expect(
      User.find(user.id).last_logged_in_at
    ).to eq(time)
  }
end
```

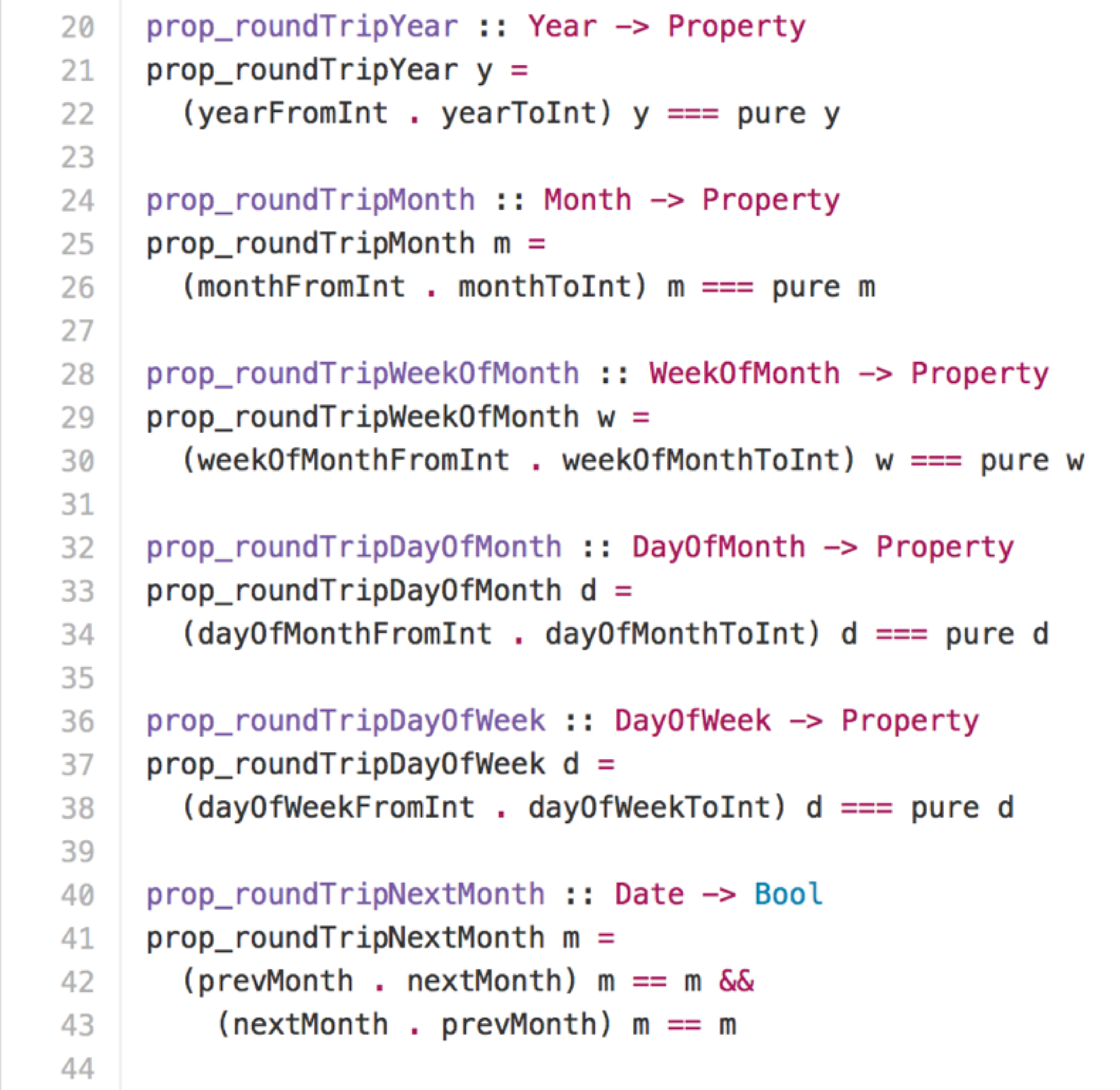
```
it "can round-trip last-logged-in" do
  property_of {
    (Time.current - float.abs)
  }.check { |time|
    user = User.create(
      username: "Sam",
      last_logged_in_at: time,
    )
    expect(
      User.find(user.id).last_logged_in_at
    ).to eq(time)
  }
end
```

```
it "can round-trip last-logged-in" do
  property_of {
    (Time.current - float.abs)
  }.check { |time|
    user = User.create(
      username: "Sam",
      last_logged_in_at: time,
    )
    expect(
      User.find(user.id).last_logged_in_at
    ).to eq(time)
  }
end
```

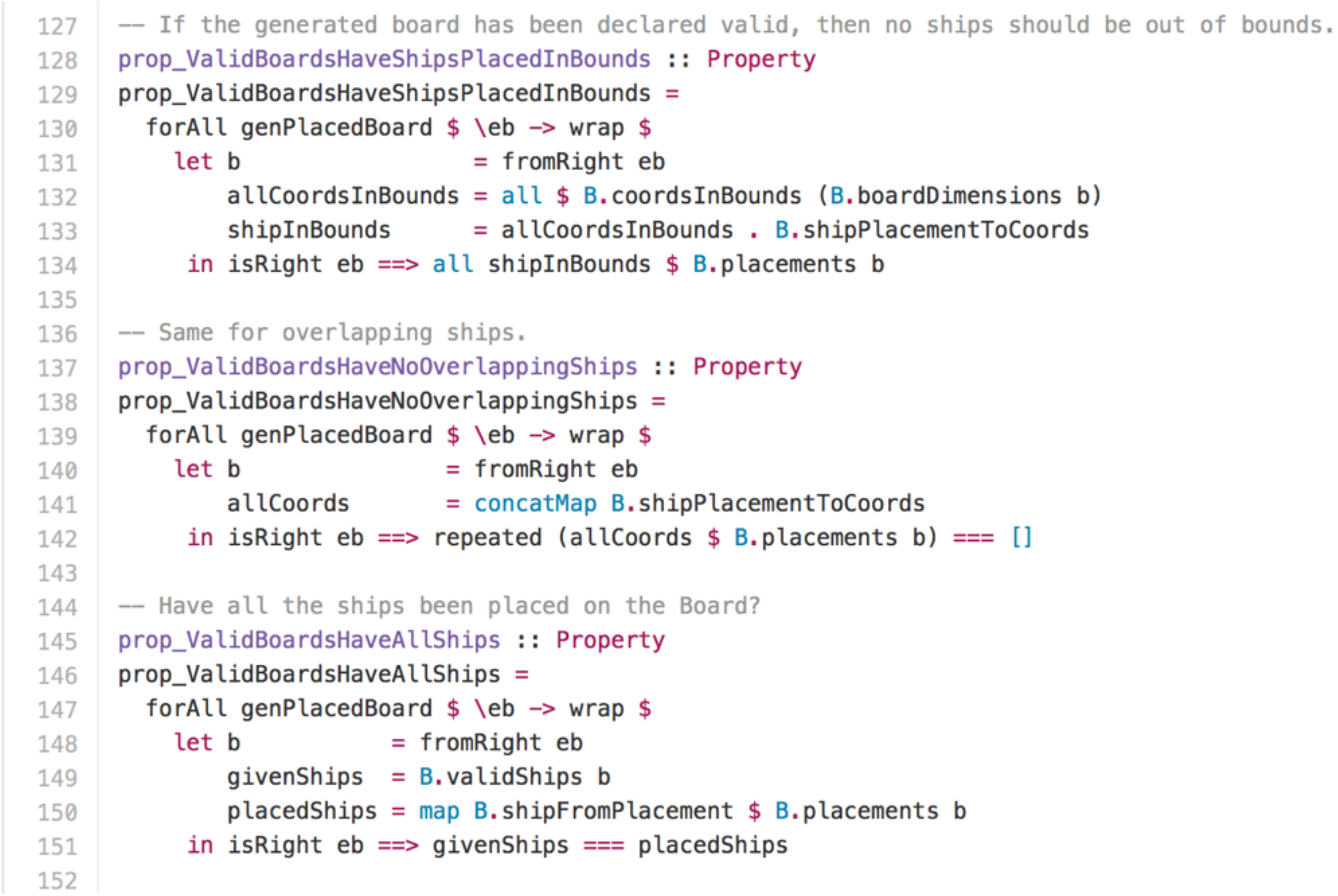
```
failure: 0 tests, on:
Sat, 13 Jun 2015 04:39:52 UTC +00:00
    can round-trip last-logged-in (FAILED - 1)

Failures:

  1) User round-trip can round-trip last-logged-in
     Failure/Error:
expect(User.find(user.id).last_logged_in_at).to eq
time

       expected: 2015-06-13 04:39:52.835645641 +0000
            got: 2015-06-13 04:39:52.835645000 +0000
```

```
it "after_transition args" do
  property_of {
    array { choose boolean, string, integer}
  }.check { |args|
    test = -> (a) { expect(a).to eq(args) }

    machine = Class.new do
      state_machine initial: :stopped do
        event :go do
          transition :stopped => :going
        end
        after_transition(:stopped => :going,
                         :do => proc { |machine,transition|
                           test.call(transition.args)
                         })
      end
    end

    machine.new.go(*args)
  }
end
```

```
failure: 1 tests, on:
["y}K'ID", "aR/-xm", "^H:/_B", true, false, true]
found a reduced failure case:
["y}K'D", "aR/-xm", "^H:/_B", true, false, true]
found a reduced failure case:
["}K'D", "aR/-xm", "^H:/_B", true, false, true]
...
found a reduced failure case:
["", "", "", true, false, true]
found a reduced failure case:
["", "", "", true, false]
found a reduced failure case:
["", "", "", false]
found a reduced success:
["", "", ""]
minimal failed data is:
["", "", "", false]
```



```haskell
prop_roundTripYear :: Year -> Property
prop_roundTripYear y =
  (yearFromInt . yearToInt) y === pure y

prop_roundTripMonth :: Month -> Property
prop_roundTripMonth m =
  (monthFromInt . monthToInt) m === pure m

prop_roundTripWeekOfMonth :: WeekOfMonth -> Property
prop_roundTripWeekOfMonth w =
  (weekOfMonthFromInt . weekOfMonthToInt) w === pure w

prop_roundTripDayOfMonth :: DayOfMonth -> Property
prop_roundTripDayOfMonth d =
  (dayOfMonthFromInt . dayOfMonthToInt) d === pure d

prop_roundTripDayOfWeek :: DayOfWeek -> Property
prop_roundTripDayOfWeek d =
  (dayOfWeekFromInt . dayOfWeekToInt) d === pure d

prop_roundTripNextMonth :: Date -> Bool
prop_roundTripNextMonth m =
  (prevMonth . nextMonth) m == m &&
    (nextMonth . prevMonth) m == m

```



```haskell
-- If the generated board has been declared valid, then no ships should be out of bounds.
prop_ValidBoardsHaveShipsPlacedInBounds :: Property
prop_ValidBoardsHaveShipsPlacedInBounds =
  forAll genPlacedBoard $ \eb -> wrap $
    let b                 = fromRight eb
        allCoordsInBounds = all $ B.coordsInBounds (B.boardDimensions b)
        shipInBounds      = allCoordsInBounds . B.shipPlacementToCoords
     in isRight eb ==> all shipInBounds $ B.placements b

-- Same for overlapping ships.
prop_ValidBoardsHaveNoOverlappingShips :: Property
prop_ValidBoardsHaveNoOverlappingShips =
  forAll genPlacedBoard $ \eb -> wrap $
    let b               = fromRight eb
        allCoords       = concatMap B.shipPlacementToCoords
     in isRight eb ==> repeated (allCoords $ B.placements b) === []

-- Have all the ships been placed on the Board?
prop_ValidBoardsHaveAllShips :: Property
prop_ValidBoardsHaveAllShips =
  forAll genPlacedBoard $ \eb -> wrap $
    let b           = fromRight eb
        givenShips  = B.validShips b
        placedShips = map B.shipFromPlacement $ B.placements b
     in isRight eb ==> givenShips === placedShips
```

Random

vs

Exhaustive

Refs, Credits and Things to Look At

- fsharpforfunandprofit.com (Property-Based Testing Posts)
- github.com/charleso/property-testing-preso (LambdaJam talk)
- Rantly (Ruby, used in examples)
- QuickCheck, SmallCheck (Haskell)
- Hypothesis (Python)

Fin.

Rob Howard
@damncabbage
robhoward.id.au